Technical Manual

Ultrasonic Cat Softener

Group Cassia

MTRX 3700 Mechatronics 3  
Major Project 2009

**Note!!! Everything in red is a comment or placeholder, and MUST be deleted or replaced.**

**You should also delete sections that are not appropriate to your project.**

Date : 4 November 2009

Revision : 0.1

Prepared by : Group Cassia:

Distribution : MTRX 3700 Lecturer (David Rye)

# Introduction

## Document Identification

This document describes the design of something. This document is prepared by Group ZULU for assessment in MTRX 3700 in 2100.

## System Overview

Cassia talking scales is a portable microprocessor-controlled scale designed to be used by the average consumer with ease of use for the visually impaired. It can be modified by the client to display in either grams or ounces with a weight limitation of 1kg load (1000grams, 35.2739 ounces) at any given time.  
  
  
All user interface is driven by a keypad list interface. Options are listed and results displayed on an LCD screen as well as spoken by the scales through the integrated speaker.Document Overview

## Reference Documents

The present document is prepared on the basis of the following reference documents, and should be read in conjunction with them.

“PIC18F452 Data Sheet”  
“Winbond TTS Manual”  
“Altronics Keypad interpereter”  
"MM74C922 16 Key Encoder Datasheet "  
"ALTRONICS Z7000A LCD Module Datasheet"  
"Samsung KS0066U LCD Controller

### Datasheet Acronyms and Abbreviations

lists the acronyms and abbreviations used in this document.

Table -: Acronyms and Abbreviations.

|  |  |
| --- | --- |
| **Acronym** | **Meaning** |
| ACFR | Australian Centre for Field Robotics |
| CTS | Continuous *or* Continuously |
| PDL | Program Design Language |
| TBD | To Be Done, *also* To Be Defined |
| TLA | Three Letter Acronym |
| USYD | The University of Sydney |
| PIC | PIC18F452 |
| LCD | Liquid Crystal Display |
| MPU | Microprocessor Unit |
| A/D A2D | Analogue to Digital Converter |
| ASCII | American Standard Code for Information Interchange |

# System Description

This section is intended to give a general overview of the basis for the something system design, of its division into hardware and software modules, and of its development and implementation.

## Introduction

Give a technical description of the function of the *whole system*, in terms of its constituent parts, here termed *modules*. Generally, a module will have hardware and software parts.

### Voltage Measurement Module

The scales hardware is a cantilever beam approximately 30cm long and 6cm wide. When masses are added to a pan on the end of the beam, the deformation of the beam is measured by 4 strain gauges located near the fixed end of the beam. The resistance of these sensors is measured, and then normalised to a 0-V++ range by an op-amp.   
The voltage produced is measured by the A/D converter on the PIC (PORTE<1>), and recorded for further processing at a later time. The number saved is a 10-bit binary number, with 0 representing 0V, and 1024 representing V++ (negative voltages will be interpreted as 0V).  
V++ is generated by a variable voltage divider circuit. This can be manually adjusted using two trimpots (coarse and fine), to generate a voltage between 0 and 5V. By default this is set to 3.7V, allowing the full range of 0-1000g. If the scales hardware changed or needed to be adjusted, this enables an easy way to keep the system performing to specification.

### Calibration and Voltage to Weight Conversion

Based upon a linear gradient and intercept that is set during the calibration mode, the function getWeight() will look at the last voltage stored and convert it to a value of either grams or ounces (using a separate linear equation for the desired unit). Once converted, the N most recent values of mass are saved to a circular buffer, which is averaged and then converted into a string of characters to be displayed on Hyperterm or the LCD.  
  
Calibration mode allows a technical user to adjust the accuracy of the scales. After entering factory mode, the user may enter calibration mode. The scales will record the voltage generated by the scales when there is no mass on it, and also when some non-zero mass is placed on the scales. Using the serial receive capability via hyperterm, the user is prompted to enter the known mass on the scales, which is interpreted as a number. These two datapoints are passed to the calibrate() function, which will generate a new gradient and x-intercept that will be used for future weighings.

### String Storage and Retrieval

There is a very limited amount of available RAM on the PIC, so most of the strings that are used by the text to speech, LCD display, and the serial transmission functions are stored in the FLASH memory until they are ready to be used. When a function wants to send one of these strings, they access it via the function "get\_rom\_string(unsigned rom char\*)", which will fetch the sequence of characters via the TLATCH register, and then copy the string to a static location in local memory where other parts of the program can send it as if it were simply available in RAM. Since this location is shared by many functions, these values are copied to one of the output streams as soon as it is possible, because there is no guarantee that this temporay string will not have changed.

## Operational Scenarios

Describe how the system is to be used. There may be several different ways that it can be used, perhaps involving different users, or classes of user. Present use case diagrams here if you are using them. Each operational scenario is a part through a use case diagram – a way of using the system, with different outcomes or methods of use.

You should also consider the various failures that may occur, and the consequences of these failures.

## System Requirements

The operational scenarios considered place certain requirements on the whole something system, and on the modules that comprise it.

Statement of requirements that affect the system as a whole, and are not restricted to only a subset of its modules.

## Module Design

## Module Requirements: Module X

### Scales and Voltage measurement

#### Functional Requirements

##### Inputs

The scales require two inputs: a +5.0V input voltage (which serves to power the active components on the board as well as a reference voltage), and a GND/0V reference signal, which should match the GND voltage of the A/D converter.

The voltage divider circuit also requires a +5.0V and 0V input.

##### Process

Further details about the operation of the scales can be found by reading the “Strain Gauge Amplifier Design Notes” (see references).

The voltage divider will generate a fixed voltage between 0-5V, depending on how it is adjusted.

##### Outputs

A single output voltage is produced from the scales, which will range between 0V and 5V.

The voltage divider outputs a fixed voltage of 3.8V as a default.

##### Timing

N/A

##### Failure modes

Over the lifetime of the equipment, it is likely that the output voltage of the scales will 'drift' up or down as different components degrade at different rates. To account for this, the scales can be adjusted using the trimpots on the board (see Strain Gauge Design Notes) OR the Vref provided to the A/D converter can be adjusted using the variable voltage divider circuit.

#### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

##### Performance

The hardware of the scales will function unpredictably in high temperatures (above 70C). In order to guarantee a predictable output voltage, care should be taken to provide a steady +5V at the input. Strong oscillating magnetic fields may cause induced voltages in the output lines.

##### Interfaces

The output channel from the scales needs to be connected to pin A<1> on the PIC in order for the A/D module to interface with the signal.

The reference voltage from the voltage divider must be connected to pin A<3>.

### The A/D Converter system is a core part of the system, as it is used to obtain readings from the scales hardware.

#### Functional Requirements

The functional requirements of the A/D module are particularly concerned with taking samples at correct intervals. This is important to guarantee constancy and reliability of the output.

##### Inputs

The A/D system requires a 5 V supply voltage to operate. The input signal must be an analogue voltage on one pin between 0 and 5 V.

##### Process

The module must operate fairly rapidly, as its operation will be performed quite frequently. A low computational complexity is highly desired.

##### Outputs

The output of the A/D converter must be a 10 bit digital value, stored in a 16 bit integer. This value must be a consistent representation of the input voltage. An input voltage of 0 V should output 0, and an input voltage of 5 volts should output 1024, thus giving a full range of possible values.

##### Timing

The sampling rate must be strictly timed such that it is regular and predictable. A suitable frequency is to be chosen, and the A/D converted must take samples at a rate as close to that frequency as is feasibly possible.

#### Non-Functional Requirements

The technical nature of the A/D module means that there are the non-functional requirements are strictly secondary to functional requirements. However, during the design process several possible non-functional requirements were specified.

##### Performance

Whilst the performance of the real-time part of the module is a functional requirement, any other sections can, theoretically, take some time to execute. However, it would be preferred if computational complexity for these parts is at a feasible minimum.

##### Interfaces

Ideally the A/D system should be entirely modular. Unfortunately, the complicated nature of the system makes this unlikely; see below in Design Constraints.

##### Design Constraints

The design of the microcontroller is such that it makes it quite difficult to build the system in a completely modular fashion. In particular, the limit to only two distinct interrupts means some parts of the system that would be preferably separated cannot be. In addition, the complex real-time nature of the system makes it somewhat difficult to completely separate the A/D functionality, which is a core part of the program, from other sections.

### Calibration

The calibration is an important part of ensuring the accuracy of the entire system, and is required to be able to accurately adjust the way the program converts the raw voltage from the scales strain gauges into numerically represented mass.

#### Functional Requirements

The main function requirement of calibration was accuracy of results, since they would directly relate to the accuracy of the entire system.

##### Inputs

The calibration needs to receive voltages and masses as input, in order to create a conversion equation. Unfortunately, these inputs are prone to being highly inaccurate. The strain gauges supplying the voltage are greatly affected by vibrations and oscillations, and the input masses themselves can be inaccurate. Additionally, a mass entered by the user can be inaccurate, either because of their own error in weighing, or if not enough significant figures are entered (particularly a problem when dealing with ounces).

##### Process

The internal process of the calibration module has to be as accurate as possible. Accuracy to many significant figures is preferred over integers.

Ideally, the module’s calculations should be able to produce an accurate calibration, which will produce a weight value with a maximum of ±1 g error, and it should be able to do so for any mass from 1 g to 1 kg.

##### Outputs

The output of the calibration module should ideally be as accurate as possible. Floats are preferred over integers, because this will enhance the accuracy of other parts of the program. Additionally, it must be able to output in both ounces and grams, no matter the input unit.

#### Non-Functional Requirements

Since the calibration module is mostly concerned with supplying accurate data to other parts of the system, the non-functional requirements are less important than the functional. However, there are still several aspects of non-functional requirements, which were considered in some depth during design of the module.

##### Performance

The computational complexity of the calibration module was a minor consideration because it would only be performed occasionally, and with not in real-time.

##### Interfaces

Simplicity for the user was an important issue. Whilst it would be possible to get a quite accurate system by creating a table of all possible voltage and mass combinations up to 1 kg, this would be an extremely difficult system to use. Ideally, the system should be able to produce a decent calibration with just two distinct voltage and mass readings.

##### Design Constraints

The limitations of the C programming language and the PIC microcontroller meant that extremely complicated calibrations were out of the question. It is difficult to perform complicated calculations, and too many calculations lead to a variety of errors, including rounding errors. The best solution to the calibration problem would likely be an n-degree polynomial line of best fit, however doing this using the tools available is quite impractical.

### Serial Transmit

#### Functional Requirements

This section describes the functional requirements of the Serial Transmit module – those requirements that must be met if the module (and system) is to function correctly.

##### Inputs

The input into the serial transmit module is encoded as a string of ASCII characters terminated by a null character (‘\0’, hex value 0x00). These strings may be dynamically generated by the program, as with numerical mass values, or defined prior to run-time, as with menu displays.

There is no correction or sanitisation of input. The module will transmit whatever string is passed to it.

The requirements for correct operation are:

* that no single string is longer than the size of the transmit buffer, which is 64 characters long;
* that many shorter strings are not transmitted immediately after each other, thus exceeding the size of the buffer with the cumulative length of all of the strings added together.

##### Process

The module must copy the input string into a circular buffer, ending when the null-termination character is reached. It must then copy one character at a time out of the buffer into the transmit shift register.

##### Outputs

The serial transmit module outputs ASCII characters over an RS232 serial connection, into the HyperTerminal terminal emulator via a DB9 male-to-male straight-through cable.

##### Timing

Timing is specified by the RS232 protocol. The connection parameters are as follows:

|  |  |
| --- | --- |
| Baud Rate | 9600 |
| Message length | 8 |
| Parity | None |
| Stop bits | 1 |

##### Failure modes

No specification was made about maintaining functionality during hardware failure. It is worth noting that the system is dual-input, dual-output, so output can be provided via the LCD module if serial transmit becomes non-functional.

#### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

##### Performance

It is necessary to consider timing in sending a string to the transmit module. The amount of time required between transmit operations obviously depends on the string length. An enforced delay of 10,000 clock cycles is provided in the module. Further delays may be necessary if transmitting multiple strings of moderate size (greater than about half the buffer length). If sufficient delays are not provided, basic functionality will be retained, but the strings may be badly formed with the loss of some characters due to overwriting. Testing with the planned strings is the best method to ensure correct functionality.

##### Interfaces

It is desirable for the serial transmit module to be able to be passed a simple null-terminated string, as this is the common format used for data storage in other parts of the program..

##### Design Constraints

The processor clock speed of 4 MHz is relatively low, meaning that the module cannot afford to poll a bit to determine when each character has been sent, and must use the transmit interrupt instead. RAM is also severely constrained, restricting the length of the transmit buffer. The software part of the module must be programmed in the ANSI C language.

### Serial Receive

#### Functional Requirements

This section describes the functional requirements of Serial Reception – those requirements that must be met if the module is to function correctly.

##### Inputs

Inputs into the serial reception module are ASCII characters transmitted over an RS232 serial connection. These inputs are made in the hyperterminal terminal emulator via a DB9 male to male straight through cable.

##### Process

The characters received are stored in a circular buffer and are progressively parsed in the program’s main loop.

##### Outputs

The output produced is a command character which is parsed to change the system’s settings.

##### Timing

Timing on the serial receive module is specified by the RS232 protocol. The connection parameters are as follows:

|  |  |
| --- | --- |
| Baud Rate | 9600 |
| Message length | 8 |
| Parity | None |
| Stop bits | 1 |

##### Failure modes

In case of failure, inputs are still possible on the local input device – keypad.

#### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

##### Performance

Reception of a character via the serial protocol should launch the system into USER\_REMOTE mode.

##### Interfaces

The user interface provides updating information on what characters have what effect in the current mode.

##### Design Constraints

So as to only deal with inputs when they become available, the Serial Reception modules operates using the RC interrupt. The received character buffer is intentionally controlled at a low volume to conserve RAM.

### Keypad Module

#### Functional requirements

##### Inputs:

* The user inputs a button on the keypad that corresponds to a desired functionality for the talking scale
* Depending on which key was triggered by the user, it will send a high signal to the corresponding row/column pins to the encoder chip
* The encoder chip will determine which key was pressed and send a 4 data output signals along with the ‘data available’ signal ( used to trigger an external interrupt) to the system where it will be processed the received data for further functionality purposes such as menu mode changes.

##### Process

* The process that is involved within this module mainly occurs in two main locations-
  + The encoder chip where it will receive the input from the user via the keypad and it will process the data depending on which row and column was trigger and send out 4 data signals along with the trigger signal according to a lookup table
  + The talking scale system after it receives the data from the encoder chip, the main system will process the data and see which functionality is required by the user and will set a flag corresponding to the desired option.
* All of this happens within a high priority external interrupt

##### Output

* The outputs from the keypad are essentially similar to the input of the module depending on which stand point you are looking at
* Essentially the output of the module is the 4 data signals coming out of the encoder chip which can be used to determine which key was pressed by looking at the lookup table
* The other required output is through the ‘data available’ signal which is set to high when a key is pressed and is used to set off a interrupt within the PIC board.

##### Timing

* Timing is a considerable factor within this module as there two areas where timing is essential :
  + The key bounce and key scan rate where it is simply controlled by connecting a capacitor across the Cose and Cmsk of the encoder chip. The timing of the scan and key bounce rate is controlled depending on the value of the capacitor as long as it gives at least 10ms delay which should be satisfactory.
  + The interrupt firing should be quick to process the required functionality

#### Non functional Requirements

##### Performance

* Requires that the keypad to be fully functional and detect every key press without much delays
* Be able to detect the right key press each time
* Be as quick as possible to detect each individual key press

##### Interface

* The keypad should interface with the LCD for menu functionality purposes
* Menu options should be displayed clearly for users to see if the keypad is functional when moving through the different menu options.

##### Design constraints

* Had to implemented through C programming
* Had to interface with the PIC18F452 board
* Restrictions to pins allocation
* Hardware restriction with size of the system box

### LCD Module

#### Functional Requirements

In order for the LCD to work properly, it must be powered on to 5V DC supply voltage and initialised with the proper timing between instructions. The 8-bit interface initialisation mode has to follow the condition of fosc = 270 KHz. The display contrast adjustment voltage must not exceed 5V. It must have an input voltage otherwise it won’t show anything on the screen.

##### Inputs

The module accepts data from other modules by using the string\_LCD function. This function stores a string that is going to be send to the LCD in the circular buffer. The input string needs to be properly formatted, using carriage return and line feed characters to indicate line changes as specified, and having lines of no more than, and preferable equal to, 16 characters.

##### Process

The module uses a circular buffer to store input such that it can be output character by character when the LCD hardware is ready. After checking that the LCD is not busy, a single character can be sent. This process is repeated until all available characters are sent.

##### Outputs

The output of the module should be a two line display on the LCD, containing the input text. Based on the carriage return and line feed characters in the input string, the input should be appropriately written on the correct rows of the screen.

##### Timing

The initialisation of the LCD must be performed in a way that adheres to the strict timing required by the hardware. These timings can be found in the Samsung KS0066U LCD driver & controller datasheet.

#### Non-Functional Requirements

Since the LCD module involves no user input, and its output is entirely governed by the user interface module, the only non-functional requirements are to do with system integration.

##### Performance

Ideally, the LCD module should spend the minimum feasible time in both storing an input message into the buffer, and sending the characters to the LCD. Since the LCD module is used quite frequently during program execution, this will improve the overall performance of the whole system.

##### Design Constraints

The microcontroller C environment is poorly equipped for delicate string manipulation, making it difficult to validate input strings. Additionally, the limitations of C meant the circular buffer was not as easy to use as it could have been in a higher level language, such as C++, where a proper queue or list structure could have been implemented.

### Speech Output

#### Functional Requirements

This section describes the functional requirements of Speech Output – those requirements that must be met if the module is to function correctly.

##### Inputs

When activated either by changes in the status of the unit or through dedicated input on the keypad, an ASCII text string is compiled in RAM and transmitted to the TTS chip via the SPI module of the microcontroller.

##### Process

The strings sent to the Speech Output module are converted to and audio signal by the Winbond WTS701 chip.

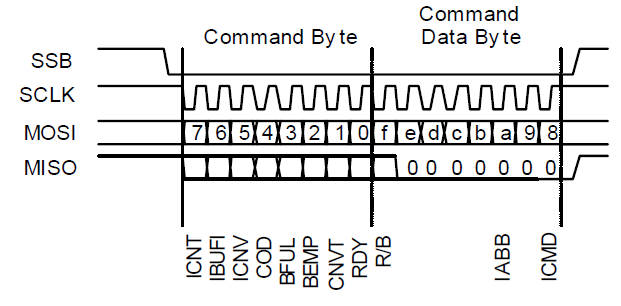
##### Outputs

The output produced is an audio signal delivered on the speaker pins of the TTS module board.

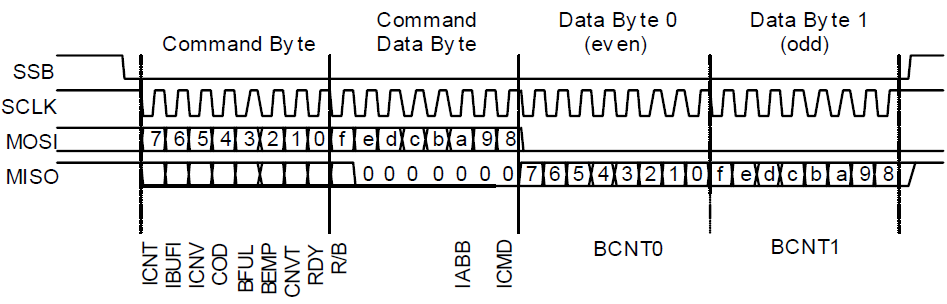
##### Timing

Communications between the microcontroller and the TTS chip must comply with the SPI protocol detailed in the Winbond WTS701 datasheet diagrammed below. The clock frequency must not exceed 5MHz.

16-bit commands:



32-bit commands



##### Failure modes

Failure of the TTS chip requires a restart and re-initialisation.

#### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

##### Performance

The speech output module should produce speech output on request and also on any change such as change in weight, mode or units.

##### Interfaces

Ideally, output strings are formatted using the TTS chip’s phonetic character set to produce better understood speech output.

##### Design Constraints

To conserve RAM, output strings are stored in program memory and copied into a buffer to be sent to the TTS chip.

### Functional Requirements

This section describes the functional requirements of Module X – those requirements that must be met if the module (and system) is to function correctly.

#### Inputs

Describe each external input, including signal encoding and timing, message encoding and timing, protocols, file formats, protection against input errors, etc, as relevant.

#### Process

Describe the internal signal transformations and/or computer processing functionality required within the module, required performance limits, and error tolerances as appropriate.

#### Outputs

Describe outputs that must be produced for the module to function correctly, including timing, frequency, protocols, etc as relevant.

#### Timing

Any required timing or latency specifications that must be met.

#### Failure modes

Required functionality (if any) in the event of failure of various nominated components.

### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

#### Performance

Requirements such as computational loop time, accuracy, etc.

#### Interfaces

Qualities that are desirable in input and output interfaces, but are not required for functionality.

#### Design Constraints

Practical or commercial considerations, such as programming languages, processor or other hardware, etc.

## Conceptual Design: Module X

### Voltage Measurement

The A/D converter on the PIC is a 10 bit device, meaning that it can only measure 1024 discrete values . In order to obtain a mass resolution of 1g with a range of 0-1000g, the A/D module on the PIC is configured so that the voltage at pin A<5> will correspond to the maximum value of 1024.

The process of getting a value into the A/D register is as follows:

* Placing a mass on the beam will cause the resistance in the strain gauges to change
* This resistance is amplified by the scales hardware, and the output is a voltage from 0-5V
* This voltage is analysed by the A/D converter on the PIC. If it is 3.8V or higher, the result stored in ADRESH and ADRESL will be 1024. If it is 0V, then the result will be 0. All other values scale linearly from these endpoints.
* Since 3.8V output corresponds to 1000g on the scales, 1 'count' in the ADRES roughly equals 1g.
* When the A/D interrupt fires (see section on A/D module), the value will be recorded and averaged to reduce low frequency noise.

### Analogue to Digital Converter

Designing the basic operation of the A/D module was trivial: the A/D module is initialized, the program starts the conversion, and then the program reads the result. However, the actual operation of the module was found to be more complex.

The functional requirements for the module specify real-time operation with regular sampling at a known frequency. The original design used a timer and a compare function to trigger an interrupt, which was then used to perform the A/D conversion.

However, it was quickly realized that this did not fulfill all the functional requirements. Performing the A/D conversion inside an interrupt was slow and clumsy, and there was no way of guaranteeing that the actual conversion would perform and sample at the correct rate. Thus, another solution was researched.

It was found that the CCP2 module in the PIC microcontroller can be used with a Special Event Trigger to automatically begin the A/D conversion on a timer compare. In conjunction with a Conversion Complete interrupt, this enabled the program to perform the time consuming conversion outside the interrupt, and sample at precisely the desired rate. This solved most of the previous problems.

In addition to this, in order to improve the speed of operation of the interrupt, it was decided to modify the system so the interrupt did not attempt to convert the value in the A/D register into an integer; it merely stored it for later conversion. By setting a flag indicating a new A/D value, it was possible to move this conversion into the mainline of the program, thus saving on computational complexity in the interrupt.

Shown below is a representation of how the two result registers, the Analogue to Digital Result High (ADRESH) and the Analogue to Digital Result Low (ADRESL), correspond bitwise with the integer result.

ADRESH

H1

0

0

0

0

0

0

L2

L3

L6

L5

L4

L7

L0

H0

ADRESL

L1

Integer result

L2

L3

L6

L5

L4

L7

L0

L1

H1

0

0

0

0

0

0

H0

#### Assumptions Made

It was always assumed that the input was correct; that the strain gauges on the scales were working correctly. See Constraints, below.

#### Constraints on A/D Converter Performance

The accuracy of the A/D converter module is highly dependent on the accuracy of the scales. If the strain gauges were incorrect, or there was some sort of hardware problem, then the A/D converter would produce incorrect results.

### Calibration

Early tests with the hardware indicated that the relationship between input voltage and mass was fairly linear. An oscilloscope was used to obtain voltages for a variety of test masses. Whilst the complexity of the beam meant that true linearity was unlikely, it was sufficiently so that it was deemed acceptable to use linear fit for calibration.

To improve accuracy and speed of calculation, it was decided that the calibration and weigh methods should go directly from the raw A/D value to the mass, rather than converting the voltage to a value between 0 and 5 V first. Thus, the following graph was produced using the same data as above, but inverting the axes and using voltages as if they were the 10 bit A/D values which the microcontroller would see.

Initial prototypes of the calibration module had a very simple method, based on the above research. Using integer arithmetic for speed, the module created a linear line of best fit using a simple equation, involving calculating gradient and y‑intercept from the two data points. However, this was quickly found to be highly inaccurate for the following reasons:

* Integer arithmetic does not provide sufficient accuracy and is prone to rounding errors
* The simple equation did not fully utilize the potential accuracy available
* Two data points were a poor representation of the entire voltage-mass relationship

The last point was deemed too difficult to fix, as a more complicated fit line would be both expensive both in computational and memory terms, and it would be highly susceptible to errors in the microcontroller C environment.

After integration of some of the modules of the system, it was found that computational complexity was far less important than originally thought. Thus, it was simple to change the calibration module to use float arithmetic.

Whilst the basic method of using a straight linear fit between two points was acceptable, it was possible to improve the accuracy of the results. By multiplying the 10 bit A/D voltage values by 64, they took up a full 16 bit integer. Whilst not useful in itself, this meant that when a (fractional) gradient is applied, a larger number remains, which is more accurate. The basic weigh function was modified to:

The gradient was used as an inverse because the value would always fall in the range (0,1). By calculating and storing the inverse a greater degree of accuracy could be achieved.

Assuming two data points, at 0 mass with some voltage, and another known mass with some other voltage, the gradient-1 and offset variables can be isolated.

It is now quite easy to use these equations to calculate *g* and *f*.  
  
Another consideration was the need to produce the results in both ounces and grams. It was decided that it was impractical to perform the whole process with different mass units, and thus the most effective method would be to convert the result. It was decided that a unit conversion function would be used. To convert the g value, it would need to be divided by the conversion factor, whilst the *f* value would need to be multiplied.

Once all of these problems were solved, it was simple to implement the module in software.

#### Assumptions Made

It is assumed that all the values used during calibration are roughly accurate. Specifically, it is assume that the known mass is entered properly. An incorrect input will likely produce an extremely incorrect output.

#### Constraints on Performance

The inaccuracy of the scales is a serious problem for the calibration. Even slight fluctuations in the voltage caused by small oscillations can throw off the result significantly. If a mass smaller than the calibration mass is weighed, it is likely to be more accurate that if a larger mass is weighed, because the interpolation is more accurate than extrapolation.

### Variance calculation

This calculates the variance inherent in the calculated mass, based on of the buffer of voltage values. This buffer forms a ‘population’ of finite size, pre-defined as VBUFSIZE = 8.

Thus the general formula for population variance is applied:

for number of samples , individual samples and mean . Here VBUFSIZE, is each voltage sample in the buffer, and is the mean voltage. This mean voltage is already calculated by a separate function so it can be easily accessed by the variance function.

The decision was made to take the variance of the voltage buffer and then convert it to a mass value, rather than taking the variance of several calculated mass values, because our setup removes the need to store a separate buffer of mass values.

Floating point arithmetic is used to provide an accurate value of the deviation, which is usually much smaller than the actual mass value. The voltage values are stored as unsigned integers, but are cast to floats for the mathematical operations to avoid truncation, then back to an unsigned integer for reporting at the end.

The variance in voltage is then converted into a variance in mass, for reporting to the user, by using the same calibrated mass-voltage conversion as for the weight. However, there is an important difference. Because the variance deals only with differences in voltage rather than absolute values, there is no need to apply an offset; only the gradient in the mass-voltage line is necessary. As an example, consider a perfectly stable voltage which thus has a variance voltage of 0. Passing this into the voltage-to-mass calculation will result in , giving a non-zero variance mass when the answer should be zero. Instead, for variance calculations the correct formula is only. This is implemented with a flag, as explained in the ‘Software design’ section.

### Serial Transmit

Serial Transmit is implemented using the PIC’s USART communications functionality. This must be correctly configured in software, a process which is described elsewhere in this manual.

One software function takes the string passed to the module and copies it into the transmit buffer. This buffer is circular and uses an incrementing insertion pointer that is reset to zero when it reaches the end of the buffer. It copies each character from the provided string until it reaches the null-termination character, which is not copied.

A second software function takes a character out of the buffer and places it into the transmit shift register, where it is automatically transmitted out by the PIC. Completing this operation triggers the transmit interrupt, indicating the system is ready for another character, and this function is called again to fetch the next one. It uses an incrementing removal pointer to index the transmit buffer, which is also reset to zero when it reaches the end of the buffer.

Transmission continues until the removal and insertion index pointers have the same value, indicating all of the string has been transmitted.

The data flow is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| String in memory:  **“Hello”** | String copied into transmit buffer   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | ‘H’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ | ‘\0’ | | In interrupt, characters copied from transmit buffer into transmit shift register:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | ‘H’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ | ‘\0’ | | ↓ | | | | | | | ‘H’ | | | | | | | Character transmitted from transmit shift register via USART to HyperTerminal. |

**Rationale**

Using the inbuilt USART functionality was the simplest method to implement. Having a transmit module that can be directly passed a string, rather than having to write one character at a time into the transmit shift register, greatly simplifies ease of coding elsewhere in the program.

**Constraints on Serial Transmit Performance**

As described above, the timing issue is the greatest constraint. Hardware errors such as inoperative cables or connections will prevent correct operation, as will incorrect setup of HyperTerminal not in accordance with the settings provided above.

### : Serial Recieve.

Serial Receive is implemented using the PIC 14F852 microcontroller’s inbuilt USART communications functionality. The dataflow is as follows:

Control Character

Hyper-terminal

RS232

USART

Input buffer

Port C pins 6 and 7 are used for serial input and output respectively.

To communicate with Hyperterminal, the microcontroller’s USART module needs to be configured for communication. RC interrupts are enabled and set for high priority.

|  |  |
| --- | --- |
| TRISCbits.TRISC6 = 0; | PortC6 is input |
| PIR1bits.RCIF = 0;  PIE1bits.RCIE = 1;  IPR1bits.RCIP = 1; | Clear RC flag.  Enable RC interrupt.  High Priority for RC interrupts |
| RCSTAbits.CREN = 1; | Enable RC |
| INTCONbits.PEIE = 1; | Enable Peripheral Interrupts |

Increment insertion pointer around buffer.

On Serial RC interrupt

Copy Serial Register to insertion pointer position

Reset flags for interrupt.

Serial RC complete

Operation of the Serial recieve module is described simply in the diagram below.

#### Constraints on Serial Receive Performance

Serial Receive is quite robust and is able to handle non operative characters.

### Keypad Module

The decision was made to implement a keypad to increase the system assessable nature given to the user. Furthermore with the use of the encoder chip i.e. the MM74C922 it improve on the signal processing of the keypad and provide extra functionalities such as the key bounce feature to prevent multiple button detection. This method was chosen over the PIC18F452 pull up resistor methods cause it was quicker and had less hardware implementation, additionally it freed up more PORT pins for other purposes.

The process of how the module will work as follows:

1. User enter desired key according to what they need the system to perform
2. Encoder chip interprets the receive key press and sends out 5 signals ( 4 data outputs and 1 data available)
3. Data available will trigger an external interrupt
4. System software will interpret what key was pressed and set a mode flag that corresponds to a certain function such as ‘count’ or ‘weigh’ depending on what the user wants
5. External interrupt flag is clear so it will be ready to receive more data.

Block Diagram

Keypad pressed by user

Encoder chip process data

Software process

Key press

External interrupt flag set?

System Shut down?

Yes

No

Yes

No

Clear External

Interrupt flag

### LCD Module

The LCD module conceptual design was based on the Samsung KS0066U LCD driver & controller datasheet and the ALTRONICS Z7000A specification sheet. The first one was extensively referenced during the creation of the software for the module, whilst the second one was of great use during the development of the LCD hardware; such as pin assignments, connections to the appropriate voltage supplies and design of the variable resistor. Peer Ouwehand’s material on [how to control HD44780-based displays](http://home.iae.nl/users/pouweha/lcd/lcd.shtml), was also followed as it gave an excellent assembly example on how to control the LCD through the HD44780 chip, which is similar in design and operation to the Samsung chip being used.

Below is a block diagram demonstrating the LCD internal structure:



Note that the A and K pins were not present on the model used.

#### Assumptions Made

The LCD assumes that input strings are formatted correctly. It also assumes that the main line of the program correctly calls the sendNext function frequently enough that the buffer is not overflow.

### Speech Output

Speech output was implemented using the Winbond WTS701 English varieties due to ease of sourcing and easy interfacing to the PIC 14F852 microcontroller using inbuilt SPI (serial peripheral interface) communications functionality. The dataflow is as follows:

Microcontroller Data String

SPI module

WTS701 TTS chip

Speaker

Audio Output

The WTS701 was mounted on a dedicated board with a voltage regulator, 74LCX244 input buffer to protect from excessively high input signals and a 74LS244 output buffer to raise output levels to TTS. The circuit design notes detailing the hardware and connectors used are available at:

http://www.aeromech.usyd.edu.au/2008/MTRX3700/Course\_Material/reference/boards/TTS/TTS%20Design.pdf

Port C pins 0-5 are connected to the X5 connector in the following way:

|  |  |  |
| --- | --- | --- |
| Line | X5 Pin | Port C Pin |
| RESET | X5-1 | RC0 |
| MISO | X5-2 | RC4 (SPI in) |
| SS’ | X5-3 | RC1 |
| SCLK | X5-4 | RC3 (SPI clock) |
| R/B’ | X5-5 | RC2 |
| MOSI | X5-6 | RC5 (SPI out) |

The WTS701 also offers an interrupt signal which, although initially included on port A pin RA1, was excluded from production as unnecessary. This is because the interrupt status can be monitored through the 16bit status response that the chip outputs.

To communicate with the TTS chip, the microcontroller’s MSSP module needs to be configured for SPI communication.

|  |  |
| --- | --- |
| TRISCbits.TRISC5 = 0;  TRISCbits.TRISC3 = 0;  TRISCbits.TRISC4 = 1; | SDO is output  SCK is output  SDO is input |
| SSPCON1bits.CKP = 1;  SSPSTATbits.CKE = 1;  SSPSTATbits.SMP = 1; | clock idle is high  Data transmitted on falling edge  Sample at end of data output |
| SSPCON1bits.SSPM3 = 0;  SSPCON1bits.SSPM2 = 0;  SSPCON1bits.SSPM1 = 1;  SSPCON1bits.SSPM0 = 0; | Set clock speed as Fosc/64 |
| SSPCON1bits.SSPEN = 1; | Enable SPI |

The Ready/not Busy signal needs to be monitored to ensure that the TTS chip can adequately respond to commands. This is done at every command send as shown in the diagram following.

Is R/B’ line high?

Load SPI buffer register to begin SPI transaction.

Poll ‘BF’ bit to check if SPI read complete

On TX command.

SPI transaction completed.

No

Yes

Increment loop count.

Is loop count = 1000 ?

RESET TTS module.

No

Yes

Read SPI buffer for response byte.

SPI exit on error.

It is critical to pull down the SS’ line for every command and raise it directly after. As the TTS chip can accept commands of any length, it is necessary to ensure that the command has finished transmitting before raising the SS’ line. For this reason two functions exist to handle speaking to the TTS module, one for commands, and another which accepts a string and frames it within the starting and terminating characters for transmission. Both functions control the SS’ line before calling the transmission routine above. The initialisation sequence is as follows:

1. Apply power.
2. Send hardware reset. This is done by setting the RESET line for at least 0.5 msec.
3. Send the ‘set clock’ command – 0x14 0x00. This should return a status of 0x00 0x80.
4. Send the ‘power up’ command – 0x02 0x00. This should return a status of 0x00 0x80.
5. Send the ‘status’ command – 0x04 0x00. This should return a status of 0x05 0x80.

Once initialised correctly, the WTS701 accepts an ASCII string and converts it to an audio signal output on the two speaker pins. The initialisations sequence is described in the data sheet for the TTS chip. INCLUDE REFERENCE TO Datasheet. Usage of the text to speech conversion capability of the module is shown below. It is necessary to note that TTS is a much slower output method than LCD or Serial Out. Therefore it is not possible to estimate when the TTS chip is ready to accept input. Writing a string to the TTS while it is converting leads to unpredictable behaviour, therefore it is better to ensure that the conversion complete interrupt flag is se before clearing it and starting the next conversion.

Conversion Complete?

Clear Conversion Int.

Send New String.

On Speech Request

Speech Request Denied

Speech Initiated

No

Yes

Strings sent to the TTS module need to be prefixed with the ‘convert’ command – 0x81 0x00, and finished with the termination character 0x1A.

#### Constraints on Speech Output Performance

The Speech Output module performance is limited due to the unpredictability of the chip’s operation. The initialisation procedure on Ready timeout has not been implemented.

Now, ***for each module***, give the outline of how it will work. In this section it is appropriate to present

* The rationale for the design decisions that were made – *why* things   
  were designed the way they were
* block diagrams,
* mathematical models and algorithms,
* data flow diagrams
* state-transition diagrams
* listings of input and output formats
* listings of message and data formats
* responses to identifiable error conditions
* responses to identifiable failure conditions

as appropriate for each module.

### Assumptions Made

State any assumptions made.

### Constraints on Module X Performance

State any constraints that may prevent the design from satisfying its requirements.

# User Interface Design

## User interface is available from three states: local, factory and remote settings. Each state uses a different input setting and level of accessibility. The User Interface is based on a simple feedback structure where the user can go in and alter states such as base weight, measurement type (grams or ounces) and so forth according to their needs. Once input command is received, it changes structure and corresponding code to go with new input and will stay that way until the system is turned off or reset. The design of user interface is broken down in options that the user can choose from based on what state they are in. The code first checks the state the user is in. After this, it checks the function the user has chosen (such as weight, tare and so forth). Each function then corresponds to a different display and speech module. New functions are cross-referenced to previously received data and then displayed. The user is also given power over the speech module. As well as being able to choose state and setting, they have the added functionality of getting the system to repeat what was said (by pressing “speak” command on the keypad) or to mute (pressing “mute” command on keypad) the system. Pressing the speak button will overwrite the mute command. Upon restart, the LCD module is cleared to display the menu back to the default setting (that is, local mode).

## Interface Design User classes are defined by local, remote and factory settings. Remote and factory settings are displayed through the hyperterminal and uses a keyboard for interface whilst the local setting uses the keypad and LCD. Local and Remote settings contain mostly the same functionality whiles factory settings allows access to more sensitive calculation procedures such as calibration and statistics.

## Changes between settings is done by the HyperTerminal. The system begins start up in the local mode. If the Hyperterminal is activated, keyboard controls can be used to switch between modes. Pressing ‘l’ will send to local mode, pressing ‘r’ will send to remote mode and pressing ‘f’ will send to factory mode. Classes of User

Interface Design: Local Setting   
**User Inputs and Outputs**   
In local mode, the user interfaces with the system via the keypad and LCD. Options available can be seen in the following figure:

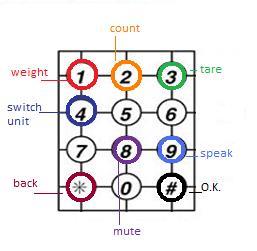


Figure 1 Keypad option layout   
1. Weight:   
Pressing this key leads to a prompt on the LCD and from the TTS to load the first mass. The module calibrates this weight according the calibration settings (see factory mode) and outputs an answer to the LCD. The TTS also received this calibration and outputs through the speaker the weight output.   
  
2. Count:   
In count mode, the first input will be to place a number of similar objects into the scale. A prompt will then take place to ask how many items are in the scales. Upon answering, the module calculates the mass of a single item. There is then a prompt for the second mode for the user to add items and calculates from previous measurements how many items are in this new mode.   
  
3. Tare:   
Tare mode allows the user to set a zero reference point. This is primarily used for adding up weights.   
  
In this mode, the user is prompted to add weights into the system. After this, they chose to ‘tare’ the value. This resets the weight scale to zero. They can then add in more weights which value will be displayed in correspondence to the zero point set by the tare.   
  
4. Switch units   
Pressing this button switches between grams or ounces as a unit of measurement output. All measurements from this point on will be displayed according to the option set here.   
  
8. Mute   
Pressing this button deactivates the speak module   
  
9. Speak   
Pressing this option repeats the last statement made by the TTS. This allows the user to rehear any instruction they may have missed.   
This button can also be pressed to deactivate the mute option.   
  
\*. Back   
This button allows the user to go back to the menu mode on the LCD   
  
#. O.k.   
Pressing this button is received as a positive command. It is pressed when a task has been undertaken and the user is ready to continue. For example, it is pressed when the user is given the prompt to add a weight and they have finished adding the weight.   
  
  
For the four main states, the LCD displays the state so that the user is made aware of what mode they are in.

## ****Input Validation and Error Trapping****

The LCD will express the state in validating the user input. Prompts from the TTS also verify the state due to the type of mode set (if not in mute). At times a key may need to be pressed twice to send to that mode.   
Operator errors are not caught in this part of the module.   
  
Interface Design: Remote and Factory Setting   
**Remote Setting: User Inputs and Outputs**   
Here the user has four options. The fifth main option, calibrate, should not be used in this mode as it’s not programmed for display on the LCD. All options are available from user input on the keyboard. Options and results are displayed through the HyperTerminal.   
  
All options are activated by pressing their first letter (e.g for ‘weigh’ option, pressing ‘w’ on the keyboard activates this state.)   
  
  
‘w’eigh:   
Weigh mode is the basic and most useful mode of the system. Upon entering this mode, the TTS and text prompts the user to place the weight they want measured onto the scale. It then calibrates this weight according the the calibration settings (see factory mode) and outputs an answer on the screen. The TTS also received this calibration and outputs through the speaker the weight output.   
  
‘c’ount:   
In count mode, the first input will be to place a number of similar objects into the scale. A prompt will then take place to ask how many items are in the scales. Upon answering, the module calculates the mass of a single item. There is then a prompt dor the second mode for the user to add items and calculates from previous measurements how many items are in this new mode.   
  
‘t’are:   
Tare mode allows the user to set a zero reference point. This is very useful for adding up weights.   
  
In this mode, the user is prompted to add weights into the system. After this, they chose to ‘tare’ the value. This resets the weight scale to zero. They can then add in more weights which value will be displayed in correspondence to the zero point set by the tare.   
  
Change ‘u’nits   
This mode changes between using grams or ounces. By choosing this option, the user can now choose to switch between the two possibilities by following the prompts. All measurements from this point on will be displayed according to the option set here.   
  
  
  
For all options, the user is made aware of what state they are in by a display in the HyperTerminal displaying the state before the result. Reading this line instantly conveys what mode is currently active.   
  
  
**Factory Setting: User Inputs and Outputs**   
Factory setting is similar to remote setting with inputs again from the keyboard. It has the same functionality as above with the additional system of ‘calibrate’, ‘statistics’, ‘raw’ and ‘samples’. These options are chosen by using uppercase letters.   
  
‘C’alibrate   
For the calibrate setting, the user is prompted to adjust the voltage output of the scales to a set weight. That is, to enter a known mass into the system. The user is then prompted to tell the machine the weight of this mass by typing the mass on the keyboard. By using the voltage output from the scales and adjusting to the input of what this output was stated to be, the program calibrates itself to use this ratio for all future measurements.   
  
‘S’tatistics   
Due to time limitations, this part of the module was not effectively completed.   
  
‘R’aw   
This option displays the ‘raw’ weight. Choosing this option is the factory mode version of ‘weigh’ from remote. Please refer to “Remote setting: user inputs and outputs >> ‘w’eigh” for more information regarding this option.   
  
Sam‘P’les   
This mode is similar to count mentioned in the other settings. Choosing this option allows the number of samples that contribute to a single weight estimate to be set.   
  
Upon selecting this mode, the user is asked to set a couple of identical weights. Once this is done and entered, the user is prompted to type on the keyboard the number of units entered. The program then calculated the weight of a single unit.   
  
The user is then prompted to empty the contents and put a new load sample in. From this sample, this module calculates the number of contents in the scale based on the previous calculation.   
  
  
  
Once again, for each option, the user is made aware of what state they are in by a display in the HyperTerminal displaying the state before the result. Reading this line instantly conveys what mode is currently active.

**Input Validation and Error Trapping**   
When a state is chosen, a fresh image on the HyperTerminal is generated with the state displayed and further options for the user to see. This tells the user if they have successfully chosen their new state.   
If an incorrect option is chosen – that is, if a letter input is made that doesn’t correspond with a letter in the menu – then nothing happens.   
If the operator chooses a wrong state, they can go back to the menu by resetting or going back by following the instructions displayed in the HyperTerminal.   
The system is case sensitive. This is needed since have two ‘c’ operations: A lower case ‘c’ is used for the count option while uppercase ‘C’ chooses the calibration mode. It is up to the user to make sure they abide by this case sensitivity in order for the system to work correctly. They will be able to tell if an error is made by the HyperTerminal showing them the option they chose.

# Hardware Design

Give a detailed description of the design of hardware. The description should include mechanical drawings, location diagrams, electrical circuit schematics, circuit simulation or test results, PCB overlays, wiring diagrams, connector pin-out lists, pneumatic/hydraulic circuit diagrams,

## Scope of the Something System Hardware

Statement of what is, and what is not, being designed and described here.

## Hardware Design

### Power Supply

The hardware portion of this module consists of a power distribution board, which takes voltage from the power supply (in this case, the +5v and GND pins from the PIC board), and provides output to the other hardware. Each output is protected by a 47uF reservoir capacitor, which provides a buffer to the rest of the system if a single piece of hardware draws a surge of current. This is especially important because the PIC processor shares the same supply rails, and would reset if it lost power even for a short time.  
The software section of this module uses the low voltage detect interrupt capability of the PIC board. In the event that the voltage to the board drops below 3V, the system will inform the user that it is about to shut down, and attempt to save the calibration and tare settings to FLASH before it runs out of power.

### Computer Design

Description of computer hardware, including all interface circuitry to sensors, actuators, and I/O hardware.

### Sensor Hardware

### Adjustable Voltage Divider

For maximum adjustability, two variable resistors were used to construct the voltage divider circuit. This makes it possible to achieve the desired voltage at Vout, even if the input voltage changes (ie, during low power operation). The value of Vout is given by:



The pre-set values (such that Vout = 3.8V) are Rv2 = 9k, Rv1 = 6k.

### Actuator Hardware

As appropriate.

### Operator Input Hardware

#### Keypad

A typical 3x4 key matrix keypad was implemented with the talking scale module with the following specifications:

* Contacts : conductive rubber
* Rating : 20mA @ 24V DC
* Contact resistance: 200 Ohms Max
* Key face dimensions : 46 x 57mm
* Life expectancy : 10 million operations per key

The following diagrams are of the wiring diagram of the keypad and the dimensions:



Figure 1. Wiring diagram of the 3x4 keypad



Figure 2. Pinouts viewed from front

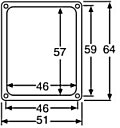


Figure 3. Dimensions of the keypad

The keypad will be implemented with an encoder chip i.e. MM74922. The purpose of the encoder chip is provide better ease of use, solves problems such as debouncing and quicker implementation. The encoder chip has the following features:

* 50 kΩ maximum switch on resistance
* On or off chip clock
* On chip row pull up device
* 2 key roll over
* Keybounce elimination with single capacitor
* Last key register at output
* 2 State output LPTTL compatible
* Wide supply range : 3V to 15V
* Low power consumption



Figure 4. connection diagrams for the MM94C922

### Operator Output Hardware

#### LCD

The Alphanumeric LCD module is powered by 5V DC supply voltage and 1mA power supply current. It communicates with the PIC microcontroller using PORTE and PORTD. The Pin Assignment used to relate the LCD to the PIC microcontroller is:

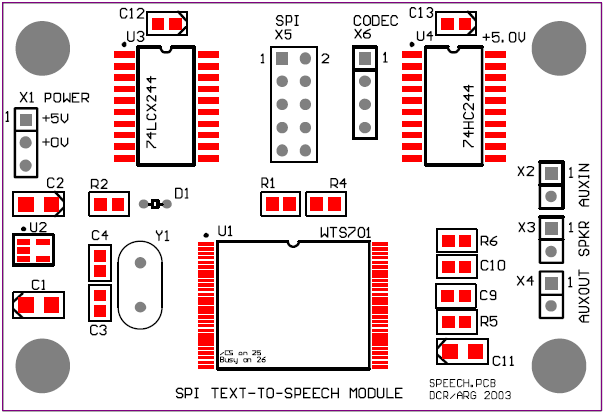
|  |  |  |  |
| --- | --- | --- | --- |
| **LCD Pin No.** | **PIC Pin No.** | **Label LCD** | **Description** |
| 1 | VSS | VSS | Ground |
| 2 | VDD | VDD | 5V supply voltage |
| 3 | N/A | VO | Contrast adjustment voltage |
| 4 | E0 | RS | Register Select signal |
| 5 | E1 | R/W | Read/Write select signal |
| 6 | E2 | E | Operation (read/write) enable |
| 7 | D0 | DB0 | Low byte data bit |
| 8 | D1 | DB1 | Low byte data bit |
| 9 | D2 | DB2 | Low byte data bit |
| 10 | D3 | DB3 | Low byte data bit |
| 11 | D4 | DB4 | High byte data bit |
| 12 | D5 | DB5 | High byte data bit |
| 13 | D6 | DB6 | High byte data bit |
| 14 | D7 | DB7 | High byte data bit |
| 15 | N/A | A | Anode |
| 16 | N/A | K | Cathode |



The Contrast adjustment voltage was controlled by using a variable resistor connected to a 5V power supply, setting this value to the sharpest display contrast. If lower contrast intensity is required, then this variable resistor can be changed to supply a lower voltage. The LCD uses a Contrast Ratio O = 10.

#### Speech Output

The Speech Output module uses the dedicated board shown below.



Full specification of the WTS701 text to speech converter chip is available in the WTS701 datasheet:

<http://www.aeromech.usyd.edu.au/2008/MTRX3700/Course_Material/reference/boards/TTS/WTS701DataSheet.pdf>

Design notes on the dedicated SPI text-to-speech board are available at:

http://www.aeromech.usyd.edu.au/2008/MTRX3700/Course\_Material/reference/boards/TTS/TTS%20Design.pdf

### Hardware Quality Assurance

Describe any measures that were taken to control (improve) hardware quality and reliability – Heartbeats, brownout conditioning/resets, reset conditions, testing and validation, etc.

## Hardware Validation

Each hardware component is tested separately using a digital multimeter and oscilloscope (as necessary) before it is integrated into the system as a whole. Due to the modular nature of the hardware design, individual components can be tested if they are suspected to be faulty and replaced individually.

## Hardware Calibration Procedures

Most of the calibration issues are solvable using the software calibration procedure (above). However, in the event that the scales are not able to measure over the desired range of masses (0 - 1000g), the operational range can be extended by using the two trimpots on the Vref supply board. This will alter the maximum output voltage of the scales (which should not exceed 3.8V when 1000g is placed upon it), and allow the PIC to measure into this range.

## Hardware Maintenance and Adjustment

For all the hardware within the project, there wasn't any heavy duty adjustment and maintenance that was required of the project. The issue that were faced in this area were:

* issue of safely storing each individual component into the system box
* each component had to carefully positioned to avoid any short circuiting or interference
* protection from dust and particles
* ventilation to prevent any overheating problems
* using the multimeter to test connections to make sure each wire is properly crimped and ready to use

# Software Design

The software requirements and overview have been dealt with elsewhere in this document. The present section addresses the design and implementation of the software that forms the X system.

## Software Design Process

From the beginning, an attempt was made to keep the design of the software layered - that is, the internal functions should behave identically, regardless of which input or output channels are being used. This was accomplished by using streams for both input and output. The reasoning behind this was that any input device (for example) could push instructions (as characters) to the input stream, and then interpreted as required by the rest of the program. The same was applied to the output: messages were streamed to output buffers, which could be transmitted/displayed/spoken at the appropriate rate.  
  
The development of software was very much from the bottom up perspective. This approach was chosen to suit the needs of the team: each team member was expected to create very small, functional, robust blocks of code that could be used and reused to build the final product. Each team member was also required to document their code on a wikispace page, which would allow other team members to understand how to use their functions, and also provide a forum for peer review and criticism.  
  
Integration was planned to start early, and be an iterative process instead of a single extensive build towards the end of the project. This was intended to allow early detection of potential problems, and also reveal shortcomings of the design that would need additional attention. However, the hardware integration was not synchronised with the software integration, and this meant that certain assumptions had to be made when putting together the software.

### Software Development Environment

Software development was done on the Microchip MPLAB Integrated Development Environment version 8.10 using the MPLAB C18 compiler (MCC18) version 3.21. See <http://ww1.microchip.com/downloads/en/DeviceDoc/51519B.pdf> for a user's guide to the MPLAB environment and <http://ww1.microchip.com/downloads/en/DeviceDoc/C18_User_Guide_51288j.pdf> for a user's guide to the C18 compiler.  
  
A Microchip ICD2 in-circuit debugger was used in conjunction with a PICDEM 2 plus prototyping/development board for software debugging and hardware development. See <http://ww1.microchip.com/downloads/en/DeviceDoc/51331C.pdf> for a user's guide to the ICD 2 debugger and <http://ww1.microchip.com/downloads/en/DeviceDoc/51275d.pdf> for a user's guide to the PICDEM 2 plus board.

### Software Implementation Stages and Test Plans

Describe the way you went about implementing the software - staged implementation, pseudocode (PDL), unit testing procedures, integration testing, etc. Identify dependencies – e.g this had to be done before that, etc.

## Software Quality Assurance

A free technical support hotline is available 24 hours a day, 7 days a week.

## Software Design Description

### Architecture

Describe the high-level architecture of the software – that is, the top-level flow of control, and how the various functional modules communicate.

In this section, you can put state transition diagrams, sequence diagrams, etc.

### Software Interface

Describe the public interface of each software module.

### Software Components

#### Numerical User Input

For two modes: Count mode and Calibrate mode, the operational state of the input devices needs to change to allow numerical data to be received. This is accomplished by setting the global flag "num\_input" to TRUE. This will change the way that key presses from the keypad are converted to characters on the input stream. For example, normally pressing the keys "1" and "4" would result in the characters "w" and "u" being placed on the input stream. However, if num\_input = TRUE, then the numerical values of the keys will be placed on the stream. By keeping track of a pointer to the location on the input stream before the user began entering numbers, this string can then be null terminated by an OK/enter, and then sent to the function "atoi()" to extract the value.

#### Count mode

Count mode is essentially an extension of weigh mode. A known number of objects are placed on the scales, and the user then tells the program how many are present. The average weight of each object is simply calculated by dividing the total weight by the number entered by the user. Then, the user is prompted to place an unknown number of identical objects on the scales. The software will then divide the total mass of the unknown items by the previously calculated weight of an individual item - yielding the number of items.

#### Analogue to Digital Converter

##### Interface

After power up, the A/D converter hardware must be initialized. This can be performed using the initialise\_AD\_converter method in the Setup module. Once this is performed, and the global interrupts enabled, the A/D converter should automatically start sampling at a frequency of 10 Hz. To get the A/D result, the newADValue flag should be checked. If it is set, then it should be cleared, and the A/D voltage result can be obtained by calling the getVoltage function.

##### Components

There are three important functions in the A/D converter module: the initialization, the interrupt, and the getVoltage method.

##### initialise\_AD\_converter

This method primarily sets all the registers required to make the A/D convert module work. It starts by enabling the A/D module, and setting it to Fosc/8 timing. Then, pin A1 is chosen as the input pin. This was used rather than pin A0 because on the PIC evaluation board a potentiometer is connected to pin A0. This pin is also configured to input. The A/D converter is set to 1 analog channel with VDD and VSS references. The output registers are set to be right justified, such that it is easier to extract and use the whole 10 bit result value.

Timer 3 is enabled and set to use a x8 prescaler. This is important for the operation of the CCP2 Special Event Trigger which starts the A/D conversion. The CCP2 module itself is initialized, and set to the Special Event Trigger mode. The CCP2 compare register is set to a precalculated value to produce trigger ever 100 ms. The A/D Conversion Complete is enabled, and set to low priority.

Timer 0 is briefly enabled, and the program waits for it to time out before continuing. This is to allow the A/D converter time to start up before any conversions are attempted. Timer 0 is then disabled, and reset.

After execution of this method, the A/D converter should be fully functional. When the global interrupt flag is enabled, the A/D converter will automatically begin sampling every 100 ms, using the low priority interrupt.

##### low\_ISR

The interrupt service routine for the low interrupt is used by several modules. This section only covers its use by the A/D conversion module.

The interrupt checks whether the A/D Conversion Complete interrupt is set, and if it has been triggered. If so, it performs the actions stated below, and then resets the flag.

The A/D result registers, ADRESH and ADRESL, are saved to global variables. This fundamentally takes a snapshot of the current value of the A/D result. The newADValue flag is then set. This allows the mainline of the program to detect the new value, and extract it from the stored values using getVoltage.

##### getVoltage

This function is designed to return a 16 bit integer containing the 10 bit voltage value extracted from the A/D converter, as specified in the functional requirements.

At a basic level, it was designed to take the high character, place it into an integer, and left shift it 8 places such that it occupied the high byte. The low character was then to be added. Unfortunately, the technical limitations of the PIC microcontroller and C compiler made this more difficult.

The first problem was that the bit shifting in the C compiler did not comply with normal ANSI C standards: instead of performing bit shifting, it seemed to perform a rotation. Rather than try to correct this problem, it was deemed simpler to replace the bit shifting by a multiplication: in this case, a multiplication by 256.

The second problem was that converting from a character to an integer caused the high bits in the integer to be undefined. This was particularly problematic when attempting to add the low character in to the integer result. A workaround was found, involving copying the character into a temporary integer first, using a bitwise AND operation to clear the top 8 bits, and only then adding it to the result.

Thus, using the above design, the function adds the high character to the integer result, and multiplies it by 256. The low character is then converted to an integer, the high byte masked out, and then it is added to the result. Finally, the result is returned.

### Calibration

#### Interface The calibration methods are part of the Scales module, but are accessed primarily by an external module containing the user interface, in this case the States module. The calibration can be performed automatically by calling the calibrate function. The following values are required as parameters:

* The A/D voltage when the scales are empty
* The A/D voltage when the scales contain a known mass
* The numerical mass correlating to the previous voltage
* The units that the mass is in

The voltages should be obtained using the getVoltage method in the Scales module. The numerical mass is input by the user, and the units should be specified.

#### Components

There are two functions in the calibration module: calibration, and convert\_unit.

#### calibration

The calibration method performs the actual calibration. It requires, as parameters, the four values specified above: unsigned integers with the voltages for 0 and some other mass, a signed integer for the actual mass, and a character of 0 for grams and 1 for ounces.  
  
The method starts by calculating the difference in voltage between the upper and lower masses. If this value is 0, the function quits, as this value would result in a divide-by-zero error.  
  
The function then proceeds to calculate the calibration parameters as specified in the Conceptual Design: Calibration section. Specifically, the following equations are used:

The second equation was, for technical reasons, split into two parts: the first calculating the value, and the second applying the negative sign.

Finally, the results are converted to the non-active unit (gram -> ounces or ounces -> grams) and all the results are stored in memory.

#### convert\_unit

The convert\_unit function is simply designed to convert a mass from grams to ounces, or ounces to grams. It uses a lookup table of conversion factors to multiply the input mass. Float arithmetic is used to assure maximum accuracy.

### Variance calculation

Firstly, the variance in voltage is calculated by calling the function getDeviation.

**Function:** unsigned int **getDeviation**(unsigned int\* set, int size, unsigned int mean)

This function returns the variance of an array named “set”, with a number of entries “size” and a supplied “mean”. It does so by implementing the variance equation described in the conceptual design, using floats. A number of different variables are defined, one for each stage of the calculation process. This enables easy debugging of each operation. The value of each is checked before adding it to the progressive total, and excluded if it is unrealistically large. This removes occasional calculation errors that can give incorrect values of the variance.

The final float for variance is cast back into an unsigned integer and returned.

In execution, the function call is getDeviation(ADSamples, VBUFSIZE, aveVoltage). This indicates that the array to be used is the sample of voltage values from the ADC, which has a length of VBUFSIZE, defined as 8. The mean value is aveVoltage which is already available after being calculated by the getAverage function for use as the weight value.

The returned value is then passed through the regular getMass function, with the parameter   
forDev = TRUE. This signals to the getMass function that there is no need to apply an offset to calibrate the voltage into a mass, as explained above.

This is then reported to the user by serial transmission as part of the Statistics menu.

### Software design: Serial Transmit

There are four software components to the serial transmit module. Firstly, a function exists to configure the USART system correctly to enable transmission. Each of the two major phases in serial transmit described in the conceptual design section above is also implemented as a function. Code also also exists in the low-priority interrupt service routine to deal with the triggering of a transmit interrupt.

**Function**: void **transmit\_setup**(void)

This is called only once, at the beginning of program execution. It makes the following assignments:

|  |  |
| --- | --- |
| **Required parameter setup** | **Code to implement** |
| Pin PORTC<7> must be output to transmit | TRISCbits.TRISC7 = 1; |
| USART in asynchronous mode due to communication with PC HyperTerminal | TXSTAbits.SYNC = 0; |
| High baud rate selected to enable 9600 baud | TXSTAbits.BRGH = 1; |
| Baud rate multiplier set for 9600 baud | SPBRG = 25; |
| 8-bit transmission mode | TXSTAbits.TX9 = 0; |
| Transmission enabled | TXSTAbits.TXEN = 1; |
| Serial port enabled | RCSTAbits.SPEN = 1; |
| Transmit interrupt low priority | IPR1bits.TXIP = 0; |

Most of these parameters are self-explanatory, defined in the project specification, or explained above in the conceptual design section. The rationale for making the transmit interrupt low priority is that receiving control characters from the user was deemed more critical than rapidly sending messages.

**Function**: void **transmit**(unsigned char\* trans\_string)

This function is called once per message to be transmitted, as explained in the conceptual design section. It requires a string of unsigned characters as input. It relies on this string being null-terminated, as it uses a while(trans\_string[i] != '\0') command to check for the end of the string. It copies the string into the output buffer transbuf indexed by the insertion pointer transbufi, and resets transbufi to zero if it reaches the end of the buffer. Lastly, it enables the transmit interrupt (using PIE1bits.TXIE = 1;) to ensure that characters begin being shifted into the transmit shift register, beginning transmission of the string.

**Function:** void **transchar**(void)

This function is called once per character to be transmitted. It simply loads the next character from the transmit buffer into the transmit shift register TXREG, where it is automatically transmitted over serial by the USART hardware. The ‘next character’ is tracked by the removal pointer transbufr, which is automatically reset to zero if it reaches the end of the transmit buffer. This function also detects when to stop sending characters from the transmit buffer, which is defined when the removal pointer catches up to the insertion pointer (transbufr == transbufi). At this point, the transmit interrupt is disabled (PIE1bits.TXIE = 0;).

**Interrupt service routine** void **low\_ISR**(void)

Inside the low-priority ISR, the system checks to see if a transmit interrupt was triggered – if transmit interrupts are enabled and the transmit shift register is empty (if(PIE1bits.TXIE == 1 && PIR1bits.TXIF == 1)). If so, the system calls transchar() for the next character.

### Keypad Software Design

The keypad software module has essentially five steps:

1. **Setting up the external interrupt and registers**
   1. Enable interrupt priority as keypad will use high priority interrupt
   2. Set PORTB<1-5> as inputs
   3. In INTCON2 register
      1. Set INTEDG1 so that external interrupt will fire on a rising edge
   4. In INTCON3
      1. Set INT1IP to enable high priority
      2. Set INT1IE to enable external interrupt
      3. Clear INT1IF to initially tell system that interrupt hasn’t occurred
2. **Detecting if the interrupt flag has been set**
   1. Checking if INT1IF has been set to go into high priority interrupt routine from the ‘data available’ signal from the encoder chip
3. **Process the data from the encoder chip**
   1. Using the following truth tables(provided below)from the MM74C922 data sheet, to process which key has been pressed by checking which PORTB<2-5> has been set through right logical shifting and bit masking
4. **Transfer the receive key press to another software module to set mode flags**
   1. String value using the lookup table of ***keypad\_lookuptable[] = {1,2,3,4,5,6,7,8,9,'\*',0,'#'}*** in combination with the truth table are sent to other modules to process which mode was triggered to enter different menu options
5. **Clear interrupt flag**
   1. Clear INT1IF so that system can be ready to receive another key press.



Figure: truth tables

### LCD Software Components

***Initialise\_LCD***

This function configures PORTD and PORTE data direction registers (TRISD/TRISE) for output. It also sets the A/D port configuration bits PCFG2:PCFG0 in ADCON1<2:0> register, in order to configure pins RE2:RE0 as digital I/O.

The PORTE direction control bits for the Enable, Read/Write and Register Select (TRISE<2:0>) pins are all configured to output.

The initialization of the LCD follows the instruction method for an 8-bit interface mode. After power on, the function calls the Delay1KTCYx function for a 30ms wait.

The first instruction after the power on delay is the Function set done by calling the Instruction\_LCD function. This register selects the 2-line mode and the display off.

After that, the Delay10TCYx function is called to generate a 39 µs wait. The next instruction in the Instruction\_LCD function is Display ON/OFF Control, where the display, cursor and blink are all turned on. Following is another 39 µs wait and the Display Clear instruction, where all the display data is cleared. Finally, the Delay100TCYx function is call to obtain a 1.53 ms delay in order to perform the last instruction: Entry Mode Set. This instruction sets the cursor/blink to move to the right and increases the data address by 1 (Increment mode). It also sets the shifting of the entire display to off.

***Instruction\_LCD***

The function sends an instruction to the LCD. It starts by calling the BF\_LCD function, which checks through the Busy Flag that the LCD is ready to receive an instruction. The Register Select then selects the Instruction register and the Read/Write selects the write operation as input. The Enable signal then allows PORTD to send out the command. The function finishes by disabling the Enable signal to show that the external write cycle is complete.

***BF\_LCD***

This function detects the LCD Busy Flag (BF) before executing the next instruction. It configures the LCD data bus for input. The Register Select is set for instruction mode and the Read/Write input to Read operation. The function then polls the Busy Flag, where it enables the signal to read the data in the Display Data RAM (DDRAM) and checks if the flag is busy (=1). The function ends when the Busy flag is clear. The LCD data bus is then set to output.

***get\_DDRAM***

The function reads the value stored in DDRAM acquired from PORTD. To do this, it first configures PORTD data direction register (TRISD) for input. The Register Select selects the Instruction register and the Read/Write selects the read operation. The signal is then enabled, allowing the DDRAM address to be read, whilst setting the Busy flag to low. The enable signal is then set to low, to make sure the read cycle is complete. The function returns data bits 0-6

***putchar\_LCD***

This function sends a character to the send\_LCD function. It starts by calling the BF\_LCD function to see if the LCD is ready. It then toggles the data between the top and bottom display lines of the LCD. An input character of ‘carriage return’ (\r) triggers the LCD to switch the data to the beginning of the top display line. An input character of ‘line feed’ (\n) triggers the LCD to switch the data to the beginning of the second display line.

***send\_LCD***

This function transmits a character to the LCD. It sets the Register Select to data mode and the Read/Write to write operation. The signal is then enabled so that the PORTD data is displayed in the LCD. The signal is then disabled to complete the cycle.

***string\_LCD***

This function stores a string in the LCDBuffer to be sent to the LCD. This function employs a circular buffer, which uses a writeIndex to indicate where the next character should be written. After writing, the writeIndex is incremented and if it reaches the size of the buffer then it is set to zero. At this point, the LCDWantsToSend flag is set, indicating to other parts of the program that data needs to be sent.

***sendNext***

The function sends the next character stored in the LCDBuffer. It first checks that there is something to be sent by checking the LCDWantsToSend flag. It there is then the putchar\_LCD function is called, in order to display the characters in the buffer. To do so, this time the circular buffer follows a readIndex, which is incremented as well and as the writeIndex, if it reaches the size of the buffer it is set to zero. The LCDWantsToSend flag is then cleared.

***isBusy***

This function is similar to the BF\_LCD function. However, instead of polling the Busy Flag, it merely returns its current value.

### Speech Output

The software in the speech output unit is built from a bottom-up perspective. The diagram below shows the hierarchical model of the main functions included in this module and their levels of complexity and interdependencies.

tts\_setup

spi\_setup

tts\_reset

SPI\_comm

tts\_send\_string

tts\_send\_command

speak

tts\_is\_ready

A

B

**Application level**

**Hardware Level**

**SPI level**

**WTS701 level**

**Initialisation level**

**B is dependent on A**

As an overview, the module is built up from hardware to handle the communications protocol specified by the WTS701 text to speech converter chip. The chip is initialised into a desired functional mode. The module provides the functions speak and tts\_is\_ready as a public interface to the application and a global array outbuff to store the received values from SPI communications. The entire module is contained within a source file ‘tts.c’. A list of the functions with brief descriptions is provided below.

void tts\_setup (void) – Initialises the tts module for operation, sets to maximum volume and enables conversion complete interrupt flagging.

void spi\_setup(void) – Initialises the MPU’s MSSP module for SPI communications. Also sets pin directions.

void tts\_reset(void) – Resets the TTS.

char SPI\_comm(char sendChar) – Takes a character ‘sendChar’ and performs a single full duplex SPI transaction. Returns the input value from the slave. Also monitors timeout conditions during the communication.

void tts\_send\_command(char\* command, char\* output) - Takes a character string containing the command to be sent and an output string for return values. Performs a non length limited communication with the WTS701 chip. Controls the slave select line to frame contiguous command characters.

void tts\_send\_string(char\* command, char\* output) - Takes a character string containing the message to be spoken and an output string for return values. Prefixes the string with the ‘convert’ command and appends with the termination character. Performs a non length limited communication with the WTS701 chip. Controls the slave select line to frame contiguous command characters.

void tts\_wait(void) – A delay function used by tts\_reset and tts\_setup.

void speak(char\* outstring) – This function is used by the application to perform a text to speech conversion. When called with a RAM text string, the function will clear any conversion finished interrupts, issue a stop command and then send the string to be converted.

char tts\_is\_ready(void) – This function is used by the application to test whether a conversion is ongoing. The function returns a logical value for whether a ‘conversion complete’ interrupt has occurred and not been cleared.

## Preconditions for Software

### Preconditions for System Startup

Describe any preconditions that must be satisfied before the system can be started.

### Preconditions for System Shutdown

Describe any preconditions that must be satisfied before the system can be stopped.

# System Performance

### Performance Testing

Give the results of testing conducted to determine the characteristics and performance of the system - memory usage, loop time, system accuracy, repeatability, ease of use, etc.

### State of the System as Delivered

After the system was demonstrated there were certain areas that needed to be addressed that involved the system’s conformation to the system specifications.  
  
Firstly the system was able to measure the weight of the item in question to a respectable level however the system was at times slow and unresponsive to commands placed by the user on the feedback of the current weight. The TTS module will be temperamental during certain moments and won’t give a response and the system would need to be resettled to get it operational again.  
  
The count function was unpredictable as well as the system will sometimes count correctly and other time it will count incorrectly however it will usually be off by a small margin, this could be due to software failure in the algorithm or there was an overflow of data.  
  
Another specification requirement was for the system to be used by a blind person, which was made possible through the implementation of the TTS to talk during certain intervals in the menu options and via a quick access guide printed in Braille that tells the user which mode corresponds to each button on the keypad.  
  
Other small areas which conform to the specifications:  
Able to toggle between ounce and grams  
Able to tare  
Switch between factory and user mode  
  
On the hardware level, the system conforms to the specifications as it was:  
Light weighted  
Low power  
Circuits were integrated via the Vero boards  
  
The system would have been portable as well however due to hardware faults, this was not possible as the smaller PIC board could not be implemented and the bigger one had to be implemented for demonstration.

### Future Improvements

Present a prioritised list of improvements to be made in future releases, giving reasons for the improvement and priority rank.

# Safety Implications

According to the NSW Occupational Health and Safety Act 2000; NSW Occupational Health and Safety Regulation 2001 the following guidelines must be kept in order to maintain user safety. Team Cassia does not take any responsibility if misuse of the product as outlined here leads to damage to person(s) or to the environment.   
  
 Do not get the product wet   
It may resemble a fish tank however the electronics inside the containment unit are very sensitive and any water may and will lead to decreased functionality, short circuiting and possible harm to the user   
 Keep the product out of reach of children at all times   
This is not a toy. The casing can break creating sharp edges that young children may hurt themselves on. Also poking fingers through the gaps or opening the side containment can lead to electric shock.   
 The product should be placed in a flat dry space, well ventilated from moisture in order to maintain ideal working conditions.   
 Do not remove or alter the electronics   
Team Cassia takes no responsibility for any damage to the product that takes place due to altering the circuitry. If you believe there is a hardware problem, please call our helpline and we will send someone to fix the problem or replace the unit for you. Under no circumstance should the user attempt to fix the problem themselves for their own safety.   
 The unit should remain closed in order to ensure all electronics are safely tucked inside away from the outer environment where they may cause harm. The self contained unit is a protective and non conducting material so as long as no one attempts to open the unit, it should remain safe for the user.   
 In case of electric shock:   
o Do not attempt to touch the user   
o Make sure you have adequate grounding (such as rubber soles on you feet so that you too do not become a conducting source)   
o With a non conducting material (such as a wooden ruler), switch off the power at the source to shut down the voltage path.   
o With a non-conductive material attempt to move the user out of the current path   
o Call an ambulance if the user appears harmed in any way from the experience   
 In case of electrical fire:   
o DO NOT USE WATER TO PUT OUT THE ELECTRICAL FIRE   
o Open all windows and doors in order to ventilate the smoke – the gas is poisonous   
o If fire starts at the wall outlet, pull out the plug by the cord or turn off power at the main switch   
o Call the fire department giving them all details and telling them it’s an electric fire   
o If a CO2 extinguisher is available, use this to attempt to put out the fire (if the fire is small)   
o If the fire becomes uncontrollable, evacuate anyone in the building and await for the fire department to arrive.

# Conclusions

**Appendix A: Title**

**A.1. Subtitle**

Appendices are used for any material that is not included in the main part of the report, usually because it would be distracting to the reader.

You would normally include **manufacturer’s data sheets**, but please do NOT include them in this report.

**Supporting calculations** should also go in an appendix.

If you use **Doxygen** (www.doxygen.org) for code documentation, generate rich text format (RTF) output, open the RTF in Word, copy, and paste it into an appendix.

Please do place your **code listings** in the following 2-column section. Make sure to use a non-proportional (mono-spaced, or “typewriter”) font, such as Courier New or Andale Mono.

Place code listings here, using style “Code Appendix” The column width will fit approximately 88 characters per column.

Each file should start on a new page, or at least at the top of a new column.